# Predicting the performance of extreme-scale supercomputer networks | Scott Pakin, Xin Yuan, and Michael Lang

modern supercomputer is the Internet in a microcosm, with tens of thousands of nodes—computers not much different from the one you may be using to read this article—all hooked together via a high-speed network. However, while computers on the Internet operate largely independently of each other, supercomputers regularly harness the power of thousands to many tens of thousands of nodes at once to run a single application significantly faster than any lone computer could. Coordinating the efforts of so many nodes requires massive amounts of communication, making the design of the interconnection network critical to the performance of the supercomputer as a whole.

In this article we present technology we are developing to predict the impact of various network-design alternatives on the overall performance of supercomputing applications *before* the supercomputer is even built. This is important because a large supercomputer can easily cost tens to hundreds of millions of dollars (and in the case of Japan's K supercomputer, over a billion dollars). Being able to evaluate network technologies during their design phase helps ensure that the supercomputer will provide as much performance as possible to applications.

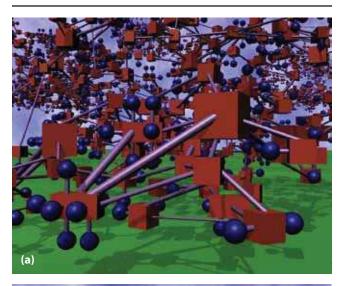
# **Network topologies**

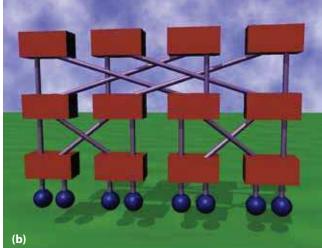
Supercomputers gain their performance edge from *parallelism*, the ability to perform many pieces of work at the same time. Taking advantage of a supercomputer consequently requires an application to divide up the work it has to perform into small chunks that can be spread over a supercomputer's nodes. In practice, some of these chunks of work depend on other chunks.

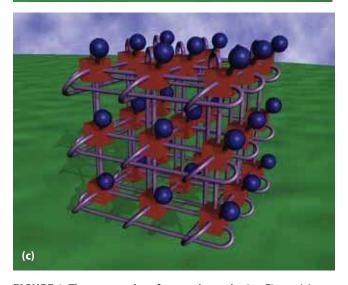
Consider, for example, an arithmetic expression such as  $(5+5)\times(6+8)$ . The two sums can be computed concurrently, but the product cannot be computed until after both sums have been computed. This necessitates communication, commonly taking the form of inter-node *messages* sent over a communication network. The node computing one sum has to tell the other node when it has finished and what sum it computed so the latter node can perform the multiplication. (Alternatively, both nodes can communicate their sum to a third node, which can multiply the two sums.) Network speed is critical to application performance. If the network is too slow relative to the time spent in computation, which is likely the case for our simple arithmetic example, there will be no performance gain to be had from parallelism, and the supercomputer's performance capabilities will be wasted.

While the Internet is composed of a motley connection of subnetworks haphazardly linked together, supercomputer networks gain some of their speed advantage by exploiting homogeneous hardware arranged into regular patterns. This avoids some nodes lying in the boondocks of the network and slowing down the entire application whenever distant nodes need to communicate with them. Figure 1 illustrates three topologies out of endless possibilities. Contrast the irregular structure of figure 1(a), which illustrates the graph nature of the Internet's topology, with the symmetry in each of figures 1(b) and (c), which illustrate two common supercomputer topologies: a fat tree and a three-dimensional torus (i.e., 3-D torus).

Nodes in the figure are shown as blue spheres. (Each of a modern supercomputer's nodes typically contains 10–100 processor cores, making a node a powerful computer in its own right.) Network links are portrayed in the figure as lavender tubes and switches are portrayed as salmon-colored boxes. A *switch* receives data on one link and, based on where the data is to be delivered, sends it out on another link.







**FIGURE 1.** Three examples of network topologies. Figure (a) shows an example of a small-world network topology. Figure (b), which depicts a fat tree, and figure (c), which depicts a 3-D torus, are two common supercomputer network topologies.

For example, if the leftmost node in the fat tree depicted by figure 1(b) needed to communicate with the rightmost node, it could send data to the switch above it, which could forward the data to the switch above it and then to the switch above it. The topmost switch could then forward the data diagonally down and to the right, then diagonally down and to the right again, and finally down to the destination node.

An alternative route would be to start with a couple of diagonally upward-and-rightward hops followed by vertically downward hops. (As an exercise, see if you can find a third path from the leftmost node to the rightmost node. Are there any more routes?) We use the term *routing algorithm* to describe the process by which switches select one route among the alternatives.

The importance of a topology such as a fat tree is that there are multiple ways to get from any node to any other node. Hence, if one route is congested, data can proceed along a different route. Consider an analogy to cars and roads, with cars representing data, roads representing links, and intersections representing switches. The more roads connecting a residential neighborhood to a commercial district, the less traffic is likely to appear on any given road. At the extreme, one could connect every node to every other node in a supercomputer to eliminate all congestion. In practice, this is not done for the same reason that there are not private roads connecting every house to every other house in a town—cost. Switches and links are expensive; hence, a network designer must simultaneously minimize the number of switches and links while maximizing the number of alternative routes between pairs of nodes. A 10,000 node supercomputer with all-to-all connectivity would require one hundred million links. At even a dollar apiece (an unrealistically small amount), this would dominate the cost of the supercomputer.

Figure 1(c) illustrates a 3-D torus, another common supercomputer network topology and one that makes different trade-offs from a fat tree with respect to switch and link count and alternative paths. In this topology, nodes and switches are arranged in a cube (or rather, rectangular cuboid) formation, and wraparound links enable data sent out one side of the network to re-enter on the other side. For example, if the node in the lower left of figure 1(c) needed to communicate with the node in the upper right, the long way would be to travel

up-up-right-right-back-back-back. However, the wraparound links enable the data to travel down to the topmost position, then left to the rightmost position, and finally forward to the backmost position, taking three hops instead of nine.

Putting cost arguments aside for the moment and assuming the same node count in both networks, could a fat tree be expected to outperform a 3-D torus, or would the 3-D torus likely be the faster network? In the next section, we discuss how to answer this question.

# Simulating networks

As creating a new network is expensive and time-consuming, we want to be able to gauge how well a given network might perform in advance of its construction. This is commonly done via network simulation—mimicking hardware's behavior with slower but vastly more malleable software. We again turn to a car-and-road analogy. Consider the situation of bumper-to-bumper traffic on two single-lane roads that merge into one single-lane road, as shown in figure 2. It would be prohibitively expensive to construct the roads and hire drivers to drive in the specified pattern just to determine the speed at which traffic can move. Instead, one could write a computer program that moves virtual cars on virtual roads and measures how much time elapses in this virtual world. In networking terms, this approach is called *flit-level simulation* because it tracks every flit (a unit of data, typically a byte) as it moves from switch to switch throughout the network.

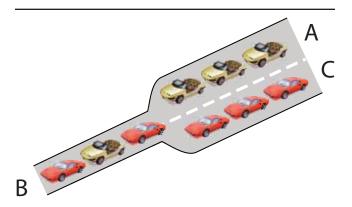


FIGURE 2. To determine the speed at which vehicles can move in bumper-to-bumper traffic on two single-lane roads that merge into one single-lane road, one can simulate this "network" using a computer program. There are different approaches to network simulation, which vary in speed and degree of realism.

At each point in (virtual) time, the simulator considers the current location of each flit in the network; the routing algorithm, which is used to decide where each flit should go next; the internal switch architecture, which controls how link contention is resolved (for example, a simple alternating of flits as illustrated by the cars in figure 2); and all the myriad other characteristics that determine performance. With regard to figure 2, a simulator would need to take into consideration not only the speed limits and layout of the road system but also the decision-making process of each driver on the road to know where the driver wants to go and how he will negotiate with other drivers as to who gets to go first when lanes merge.

There are two main problems with flit-level simulation, one inherent and one artificial. The inherent problem is that simulating a large network at such a fine level of detail is necessarily slow—vastly slower than real network hardware could run. Thousandfold slowdowns are not uncommon. In other words, the simulator might need to run for an hour to report how a network might behave over the course of a single second of execution. To put that slowdown in perspective, consider that many of the scientific applications commonly run on supercomputers at Los Alamos National Laboratory take hours to days to run; a few even require months to over a year to complete. Dilating such times by a factor of a thousand clearly limits the practicality of simulating such applications. Consequently, flit-level simulations must by necessity whittle down their inputs to a more manageable size, simulating only small networks and for only brief periods of time, which limits realism.

The artificial problem is that for simplicity of operation, simulators are typically fed synthetic communication patterns rather than communication patterns derived from actual supercomputing applications. For example, two common test patterns are uniformly random traffic in which each node sends data to some number of other nodes selected at random, and hotspot traffic in which all nodes send data to a small subset of the nodes selected at random. Second, almost all simulation studies presented in the supercomputernetwork literature assume that communication begins at fixed points in time, typically exclusively at the start of the simulation. Third, computation time is almost universally ignored, even though this can greatly affect the severity and impact of link contention.

Returning to our car-and-road metaphor, typical simulator usage would be analogous to gauging the quality of a layout of a city street under assumptions like the following:

- 1. People drive randomly from one place to another as opposed to, say, a bias to drive to the kids' school at the beginning of the day, then to the office, then to the kids' school again, and finally back home.
- 2. Everyone leaves home at exactly 9:00 a.m., drives directly to his destination, and leaves the car there. A less-common variation on this assumption is that Alice picks up Bob at exactly 8:15 a.m., Carol at exactly 8:30 a.m., and Dave at exactly 8:45 a.m. for their carpool to work—all regardless of how heavy or light the traffic happened to be at the time or whether a new highway had just been built to speed up their commute.
- 3. No one stops to work, shop, or relax; all anyone in the city does is drive.

It would be hard to lend much credence to any result of such a study, yet this is very much how supercomputer networks are analyzed today. Again, this is an artificial problem. There is no fundamental reason that such assumptions must be made; they are merely a convenience to simplify the simulation effort. In the next section, we describe how we are improving the state of the art in network-simulation technology, both in terms of simulation speed and simulation realism.

# A new approach to network simulation

Our goal is to address all of the shortcomings discussed above; in particular, our aim is to simulate all of the following:

- Full-sized applications, not synthetic communication patterns;
- Hours of application-execution time, not seconds;
- Tens of thousands of nodes, not hundreds to low thousands;
- Communication interleaved with computation, not treated as independent; and
- Communication beginning when prior communication or computation ends, not at fixed points in time.

The two mechanisms that underlie our approach are flow-based simulation and logical clocks. We now describe each of these in turn.

## Flow-based simulation

The reason that flit-based simulation is so slow is that supercomputer networks contain a massive number of components, and each of these must be simulated individually. Logically, if one were to simulate large groups of components as single entities, this would greatly reduce the amount of work, and therefore time, required to run the simulation. We therefore choose to consider a complete, end-to-end communication operation as a single unit of simulation rather than the numerous flits that get transmitted as part of that operation.

Before we explain the details precisely, we present the intuition behind our approach in terms of our running car-and-road analogy. Assuming a 40 mph speed limit and that the distance from the front of one car to the front of the next is 29 feet, the math works out to two cars per second passing any particular point on the road. Hence, if we knew that 100 cars wanted to go from point *A* to point *B* and that there was no other traffic on the road, the first car in that sequence would arrive after some given length of time (i.e., however long it takes to drive from point *A* to point *B* on an empty road, say three minutes), and the last car would arrive  $100 \div 2 = 50$  seconds later.

We now consider the variation indicated by figure 2: 100 yellow cars want to go from point *A* to point *B* at the same time that 100 red cars want to go from point C to point B. What impact does the shared segment of road have on the time it takes each of those two flows of cars to reach their destination? As before, two cars per second are reaching point *B*, but because the two flows are interleaved, only one yellow car per second and one red car per second can reach that location. The first car in each flow is not delayed, so it still takes our assumed three minutes to arrive at point B, but the last car in each flow arrives not 50 seconds later but  $100 \div 1 = 100$  seconds later.

The point of this exercise is to demonstrate that, unlike with flit-level simulation, we do not have to consider each individual car's behavior. Instead, we can analyze an entire sequence of cars at once, regardless of whether there are a hundred cars in each flow or a

million. Furthermore, we do not need to consider how the drivers negotiate the merge. All that matters is that there is an even 50-50 split between red and yellow cars on the merged segment of road, not that it went red-yellow-red-yellow versus red-red-yellow-yellow.

Our approach to network simulation works in very much the same way as the preceding analysis of traffic speeds. As in the above instance, instead of working with communication times directly, we work with communication rates, which we can easily relate back to time by noting that time = latency + (data size ÷ communication rate), where *latency* is the time it would take a single flit to move from the source node to the destination node in the absence of any other traffic. For example, suppose that the latency between node *A* and node *B* is 0.6 seconds and that all of the links between node A and node B are capable of transmitting 5 gigabytes per second. If node A were to transmit 1 gigabyte of data to node B, this communication would take a total of  $0.6 + (1.0 \div 5.0) = 0.8$  seconds.

While latencies are essentially constant and data sizes can be extracted from an application (as we will discuss further when we discuss logical clocks), communication rates vary dynamically based on the amount of link contention, the number of communications sharing a network link at any given time. Consider the network topology shown in figure 3 (i.e., a 2-D mesh).

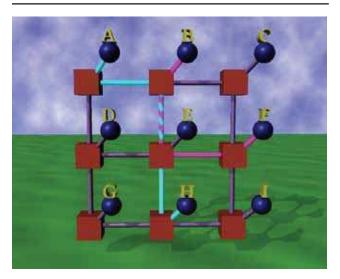


FIGURE 3. An illustration of link contention on a 2-D mesh network topology.

If node A sends data to node H via the route A-*B–E–H* (cyan links) at the same time as node *B* sends data to node *F* via the route *B–E–F* (magenta links), the *B–E* link will be shared by the two routes. Supposing the link is capable of transmitting at a rate of 5 gigabytes per second, 2.5 gigabytes per second will be allocated to each of the two communications. Because data cannot enter a link faster than it can exit, this slow link then exerts back-pressure all the way to the source nodes, slowing down the entire communication to 2.5 gigabytes per second. Using the previously mentioned sample numbers from each of the two communications will now take  $0.6 + (1.0 \div 2.5) = 1.0$  seconds instead of the contention-free 0.8 seconds computed earlier—slower but notably not twice as slow, even though the link speed effectively halved.

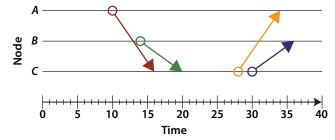
# Logical clocks

We criticized prior simulation efforts for relying on synthetic communication patterns instead of actual communication patterns derived from supercomputing applications. Our question is therefore how we can acquire an application's communication pattern so that it can be analyzed by a simulator. The enumeration of all communication that an application performs during its execution—which node sent how many bytes to whom when—is called a *communication trace*. Fortunately, intercepting and logging an application's communication operations is fairly straightforward, and there exist numerous tools for collecting communication traces.

The issue is not with *collecting* the trace but with *interpreting* it. Figure 4 helps clarify the problem. Figure 4(a) presents a trace of a communication pattern in which node *A* sent a message to node *C*, then node *B* sent a message to node *C*, then, after a brief interlude, node *C* sent a message to node *A*, and finally, node *C* sent a message to node *B*. A graphical view of this trace is shown in figure 4(b). Send and receive times are reported from the perspective of each node's clock. For example, the first line of the table in figure 4(a) indicates that node *A* reported that it sent a message to node *C* at time 10 and that node *C* reported that it received node *A*'s message at time 16.

The first problem with this type of communication trace is that supercomputer nodes seldom include pernode clocks that are globally synchronized to within half a message latency (i.e., the tolerance needed to

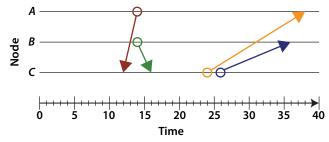
Source Node	Destination Node	Sent Time	Received Time
Α	C	10	16
В	C	14	20
C	Α	28	34
C	В	30	36



(a) Communication trace

(b) Timeline view of the trace

Source Node	Destination Node	Sent Time	Received Time
Α	C	14	12
В	C	14	16
C	Α	24	38
С	В	26	36



(c) Communication trace with poorly synchronized clocks

(d) Timeline view of the trace with poorly synchronized clocks

**FIGURE 4.** Example of a communication pattern. Figure (a) and (b) illustrate a communication trace of nodes with perfectly synchronized clocks (an unrealistic condition). Figure (c) and (d) illustrate a communication trace of nodes with poorly synchronized clocks.

avoid erroneous readings, as discussed below). These would represent a costly but rarely useful expense. Furthermore, access to a single, centralized clock would be devastating to performance—imagine figure 2 with tens of thousands of lanes merging into one. Hence, some node clocks may run slightly ahead or behind others, and even worse, some node clocks may run slightly faster or slower than others. This is called *clock drift*. Although clock-synchronization algorithms exist, software implementations are unable to synchronize clocks to a granularity fine enough to measure network-communication time.

Figure 4(c) represents the same trace as figure 4(a) but as measured with node A's clock running four time units late and node C's clock running four time units early. As the graphical depiction of this trace in figure 4(d) clarifies, the faulty clocks make the first message appear to have been received before it was sent, a physical impossibility. Furthermore, instead of each message taking a constant six time units to get from source to destination as indicated by the "perfect" trace in figure 4(a), the B-C communication in figure 4(c) appears to take only two time units while the C-B communication appears to take ten.

The second problem with using figure 4-style communication traces involves how the simulator replays the traced communication pattern. Suppose we wanted to simulate a network that runs twice as fast as the one on which the communication trace was acquired or perhaps the same network attached to processors running three times as fast as on the measurement system. It would be unreasonable in either case to expect all of the messages to be sent at the same times shown in figure 4(a). A node that receives a message sooner or finishes some computation faster may then be able to send a message earlier. We therefore do not want our simulator necessarily to simulate messages being sent at the times listed in the input trace but rather at the times that the simulated supercomputer would actually send them.

The solution to both of the preceding problems is an abstraction called a *logical clock*, first proposed by Lamport in 1978 [2] and sometimes called a *Lamport clock* after its inventor. A logical clock is a simple, integer counter that "ticks" as follows:

#### Alice's Journal

I left home at 3:30 p.m. to drive my son to soccer practice. When I got to the soccer field my watch read 3:55 p.m.

I left the soccer field five minutes later, at 4:00 p.m., to join Carol for tea. I arrived at the tea house at the same time as Carol, at 4:30 p.m. according to my watch.

After a lovely hour of tea with Carol, we both drove home at the same time, when my watch read 5:30 p.m. just in time to make dinner.

#### Carol's Journal

I left home at 6:30 p.m. to join Alice for tea. I arrived at the tea house at 7:00 p.m. according to my watch, which is right when Alice arrived.

I had an enjoyable tea time with Alice. We both left 45 minutes later, at 7:45 p.m. on my watch, to drive home. I got home at 8:05 p.m.

FIGURE 5. The ordering of events from Alice's and Carol's perspective.

- 1. When a node performs any operation (communication or computation), its clock advances its logical time by one.
- 2. When a node sends a message, its clock includes the current logical time along with the normal data.
- 3. When a node receives a message, it sets its logical clock to the maximum of its current logical time and one plus the logical time included in the message.

These rules help define a "happened before" relation (mathematically, a partial ordering) on communication operations. If one operation occurred at a smaller logical time than another, then the simulator cannot perform the second operation until the first one finishes. In contrast, if two operations occur at the same logical time, the simulator has no restrictions on the order it performs them: it can run *A* then *B*, *B* then *A*, or both simultaneously. In essence, a logical clock provides a way to globally *order* communication operations regardless of the locally observed *time* at which each operation may appear to have occurred.

To clarify using yet another driving analogy, consider Alice's and Carol's sequences of events, presented in figure 5.

In what order did those events happen? It would be incorrect to sort them by the times listed in the event descriptions because Alice and Carol may not have synchronized their watches beforehand and because

	Logical time spent at various locations			
Event	Alice's House	Carol's House	Soccer Field	Tea House
(Our story begins)	1	1	1	1
Alice's -> Soccer	1	1	2	1
Soccer - Tea	1	1	2	3
Tea → Alice's	4	1	2	3
Carol's → Tea	4	1	2	3
Tea → Carol's	4	1	2	4

Logical Time	Observable Events
1	Alice and Carol are both at home.
2	Alice is at the soccer field. Carol may be either at home or en route to the tea house. We have insufficient information to determine which.
3	Alice and Carol are both at the tea house.
4	Alice and Carol are both at home. We have insufficient information to determine who arrived first.

**FIGURE 6.** The ordering of events based on logical time.

either watch may run faster or slower than the other. Nevertheless, we can intuitively rely on what makes sense to order the specified events. Specifically, we know that Alice must have driven *to* the soccer field before driving *from* the soccer field; we know that both Alice and Carol were at the tea house at the same time; and we know that both Alice and Carol left the tea house at the same time after having tea together.

Figure 6 shows how to express that "what makes sense" intuition as formal statements of changes to logical time. The table assigns one logical clock to each *location* (as opposed to each person) mentioned and lists the events that Alice observed, in order, followed by the events that Carol observed, in order. (The results would be the same if we swapped or even interleaved Alice's and Carol's journals, as long as the events were not reordered relative to how they appear in either journal.) In our network-simulation framework, locations correspond to nodes, and a person driving from location to location corresponds to communication.

At the beginning, all locations are at logical time 1, and Alice and Carol are both in their respective home. When Alice drives to the soccer field, she must arrive some time after she was at home. The soccer field therefore increments its logical time to 2, the

maximum of its current time (1) and one plus the time at Alice's house (1 + 1). When Alice drives to the tea house, she must arrive some time after she left the soccer field. The tea house therefore increments its logical time to 3, the maximum of its current time (1) and one plus the time at the soccer field (1 + 2). When Alice drives home, she must arrive both after the last time she was there (1) and after she left the tea house (3), that is to say, at time 4.

Turning our attention to Carol, Carol must arrive at the tea house at a time later than when she was at home. However, the tea house's clock does not change because the maximum of its current time (3) and one plus the time at Carol's house (1 + 1) is already 3. Finally, when Carol drives home, she must arrive both after the last time she was there (1) and after she left the tea house (3), that is to say, at time 4.

For clarity, the bottom part of figure 6 re-sorts the data by logical time, showing which events happened at each time. From this presentation, one can infer that despite the physical times stated in the event descriptions, Alice could not possibly have returned home before Carol arrived at the tea house (time 4 versus time 3). However, the logical-clock readings in figure 6 say nothing about whether Alice arrived back at her home before Carol arrived back at her home (time 4 for

both events). More subtly, the readings do not indicate which of Alice or Carol arrived first at the tea house (as both soccer  $\rightarrow$  tea and Carol's  $\rightarrow$  tea completed at time 3); we know only that neither left (time 4) before both arrived (time 3).

Logical clocks provide an important mechanism for fulfilling the goals stated at the beginning of this section in that they enable a network simulator to reason about communication dependencies—what must happen before what—rather than physical times. One additional innovation of our network-simulation methodology is that we record computation time in physical time. In figure 6's analogy, this would be like a waiter at the tea house reporting how long Alice and Carol spent there. Maintaining this information enables the simulator to honor computation time, which may have substantial impact on communication time. Consider, for example, how much faster the cars in figure 2 would move if the yellow cars were on the road only in the morning and the red cars were on the road only in the afternoon.

Even without perfectly synchronized, drift-free node clocks, combining physical computation time with logical communication time enables us to accurately reproduce application timing measurements and provide some confidence that varying hardware parameters will lead to accurate predictions of performance. In the following section we quantify how well this works by presenting an early evaluation of our simulation methodology.

## **Initial results**

Our simulation project is still in its early stages. However, the logical-time trace acquisition software and the simulator itself are operational and support a sufficient set of features for an initial evaluation of our approach.

As a sample application, we use a hydrodynamics code developed at Los Alamos National Laboratory called PAGOSA. PAGOSA is designed to simulate high-speed fluid flow and high-rate material deformation [1]. The application comprises approximately 67,000 lines of code (about 1,000 printed pages), mostly written in Fortran but with some C. PAGOSA's constituent processes are logically arranged in a three-dimensional layout and communicate primarily

with their immediate north, south, east, west, front, and back neighbors. This is an ideal structure for a three-dimensional network such as the one shown in figure 1(c) if the application's coordinates directly map to the network's coordinates. For example, mapping a  $6 \times 6 \times 6$  PAGOSA layout onto a  $6 \times 6 \times 6$  network could be expected to perform well. In contrast, mapping it onto a  $6 \times 4 \times 9$  network would in fact make some "neighbors" not adjacent to each other, leading to link contention. In practice, users are rarely given control over the set of nodes allocated to their applications.

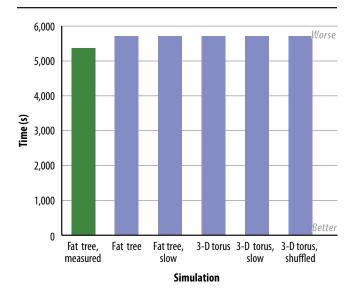
We ran PAGOSA on 1,000 nodes of a 1,600-node supercomputer called Mustang. Mustang is based on a fat-tree network such as the one shown in figure 1(b), but 200 times larger. More precisely, figure 1(b) represents what is often called a *2-ary 3-tree*, because each switch connects to two switches in each adjacent row and there are three rows of switches. Mustang uses an 18-ary 3-tree<sup>a</sup> so each switch connects to 18 switches in each adjacent row, but there are still only three rows in the network, just as in figure 1(b). As of June 2013, Mustang was rated the 137th fastest supercomputer in the world [3].

# Full-application simulation at scale

PAGOSA was configured to execute a canonical hydrodynamics test problem, the simulation of a spherical shell of beryllium being subjected from all directions to a given amount of kinetic energy, which compresses the shell. Figure 7 presents the results of simulating this PAGOSA execution using the sets of network parameters listed in table 1.

The first bar, labeled *Fat tree, measured,* indicates that the PAGOSA test problem normally takes an hour and a half to complete on Mustang. The second bar, *Fat tree,* demonstrates that our simulator is quite accurate, being only 6.4% above the correct value. Recall that our work is still in its early stages; we hope in the near future to improve simulation accuracy. The second and subsequent bars each represent between 14½ and 15½ hours of time running the simulator on a single desktop computer. This is a noteworthy success: Even though we used a thousandth of the number of nodes as in the real execution, our simulator took only tenfold the time to run. And, unlike real execution,

a. Mustang in fact contains an incomplete 18-ary 3-tree—an XGFT (3; 18, 6, 16; 1 6 18) in Öhring's notation [4]—and this is what we simulate.



**FIGURE 7.** Simulated PAGOSA execution time using the sets of network parameters in table 1.

our simulator enables limitless "what if" experimentation with different network topologies and network performance characteristics.

As a demonstration of that capability, the remaining bars in figure 7 show the results of simulating different networks from Mustang's actual network. As detailed in table 1, Fat tree, slow represents a substantially slower network than Fat tree. 3-D torus uses the same network speeds as Fat tree but with a 3-D torus topology instead of a fat tree. Likewise, 3-D torus, slow uses the same network speeds as Fat tree, slow but with a 3-D torus topology instead of a fat tree. 3-D torus, *shuffled* represents the same topology and network speeds as 3-D torus but randomly shuffles the mapping of PAGOSA processes to torus nodes. Torus networks are notoriously sensitive to process placement, and we can use our simulation technology to evaluate how sensitive a given application is to the placement of its constituent processes.

The clear implication of figure 7 is that PAGOSA's overall performance is almost completely oblivious to network performance. Despite the simulated variations in network topologies and speeds, the difference in execution time from one network to another is a tiny fraction of a percent. Although the 1,000-node run of PAGOSA communicated an aggregate of two billion messages comprising a total of 14 terabytes of data, communication time is so dominated by computation time that network speed is largely inconsequential.

## Comparison with simplistic simulators

We have shown that flow-based simulation delivers simulation speed and that logical clocks provide high fidelity to actual application execution time. The next question to consider is how our approach compares to the more simplistic approach employed by most network-simulation studies. While our simulator honors both communication dependencies and computation time, it is far more typical in the simulation literature to pretend that all messages are sent simultaneously at time 0 and to simulate the time it takes all messages to reach their destination in the absence of computation.

We configured our simulator to disregard communication dependencies and computation time, in essence dumbing down our simulator to the capabilities of a more traditional network simulator. The results, shown in figure 8, paint a very different picture of performance from figure 7.

The total height of each bar represents the time for the last message in the corresponding simulation to complete. The light purple region represents the average time for a message to complete. While figure 7 indicates that PAGOSA's total execution time is almost completely independent of communication time, figure 8 exaggerates the differences. Specifically,

**TABLE 1. Simulation parameters** 

Simulation	Topology	Link Speed (Gbps)	Switch Latency (ns)	Software Overhead (ns)
Fat tree	18-ary 3-tree	40	100	1,500
Fat tree, slow	18-ary 3-tree	10	400	4,000
3-D torus	8 x 16 x 16 torus	40	100	1,500
3-D torus, slow	8 x 16 x 16 torus	10	400	4,000
3-D torus, shuffled	8 x 16 x 16 torus	40	100	1,500

the 3-D torus requires 70% more time than the fat tree to transfer PAGOSA's two billion messages. For both network topologies, quartering the bandwidth exactly quadruples the communication time.

This study demonstrates that it is critical to include communication dependencies and computation time in a network simulation. Otherwise, differences in network topology and basic performance characteristics appear more significant than they really are. This misleading information could persuade a supercomputing center to pay extra for a faster network when a slower, less expensive network may deliver almost exactly the same performance to applications.

## **Conclusions**

Modern supercomputers are architected as vast aggregations of processors interconnected with high-speed networks. Because scientific applications are generally composed of myriad processes working together to simulate natural phenomena, communication speed is critical for efficiently coordinating all of those processes. However, engineering a high-speed network involves inevitable cost/performance trade-offs. Furthermore, all applications use the network differently, contraindicating a one-size-fits-all solution. Some applications transmit a large number of small messages; others transmit a small number of large messages. In some applications, each node communicates with only a small set of other nodes; in others, all nodes communicate with all of the others. Some applications communicate continuously throughout their execution; others alternate communication and computation phases.

Supercomputing centers want to maximize the overall performance delivered to the applications they expect to run but without overpaying for unnecessary network performance. One way to predict how well a given application will perform on a particular network in advance of its procurement is via a technique called *network simulation*. With simulation, one mimics hardware's behavior and performance characteristics using a software test bed. Simulating hardware is slower—typically many thousands of times slower—than running on true hardware but is cheap to deploy and easy to modify to investigate different design alternatives.

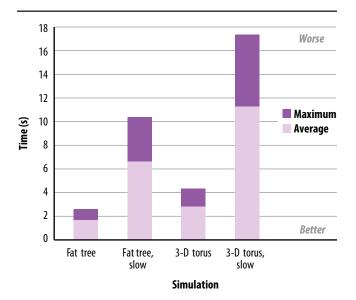


FIGURE 8. Differences in simulated communication time only.

The problem with existing network simulators and simulation studies is that they tend to incorporate so much detail that they cannot handle large numbers of nodes or substantial lengths of time. Furthermore, for simplicity of implementation they ignore the juxtaposition of communication with computation and with other communication, unrealistically assuming that all messages are initiated in a single burst. In this article we proposed addressing the speed issue with flow-based simulation and the realistic-usage issue with logical clocks that are augmented with physical computation time. To demonstrate the potential of this approach we implemented a tool to derive logicaltime traces from parallel applications and a flow-based simulator to replay those traces on different simulated network topologies and with different network performance characteristics.

One can draw the following conclusions from the experimental data we presented. First, our approach accurately simulates real execution time. Although our implementation is in its nascent stages, we already saw less than 7% error when simulating a scientific application, PAGOSA, running for an hour and a half across a 1,000-node network. Second, flow-based simulation runs at reasonable speeds. We replayed that 1,000-node, hour-and-a-half run on different simulated networks using only a single node, and it ran only 10 times slower than real time, not thousands or

tens of thousands, which is what is typical for a more traditional network simulator. Third, the common simplification of ignoring communication dependencies and computation time in network simulations exaggerates the pressure the application applies to the network and leads to incorrect assessments of network performance.

In our experiments, we found that PAGOSA performs so much computation relative to communication that the network topology and basic performance characteristics are largely inconsequential. In contrast, a more traditional network simulator would incorrectly claim 70% more performance for a fat-tree topology than for a 3-D torus topology when replaying PAGOSA's communication pattern.

In summary, combining logical time with flow-based simulation opens up new avenues for evaluating how fast applications will run on different super-computer networks, most notably supercomputer networks that have not yet been built. This capability can inform network design decisions—or even simply a selection from multiple existing networks—to help provide applications with the best communication performance that the supercomputer budget allows.

#### About the authors

Scott Pakin is a research scientist at Los Alamos National Laboratory. He has been actively working in the area of high-performance network research for over 15 years, beginning with the development of Fast Messages, one of the first high-speed messaging layers for a commodity supercomputing network, Myrinet; and more recently including the Cell Messaging Layer, which makes it practical for computational accelerators to communicate directly across a deep, heterogeneous network hierarchy; and coNCePTuaL, a domain-specific language, compiler, and run-time system that facilitate the rapid generation of custom network speed tests with repeatable results.

Dr. Pakin has served on numerous network-related national and international conference and workshop program committees, including the position of area cochair for the Architecture and Networks track of this year's annual Supercomputing Conference (SC'13) and continuing cochair service for the annual Communication Architecture for Scalable Systems (CASS) workshop. He also served as a guest editor for the

November 2012 special issue of Elsevier's *Journal of Parallel and Distributed Computing*, which focused on interconnection networks. Dr. Pakin received a BS in mathematics/computer science with research honors from Carnegie Mellon University in 1992, an MS in computer science from the University of Illinois at Urbana-Champaign in 1995, and a PhD in computer science from the University of Illinois at Urbana-Champaign in 2001.

Xin Yuan is a full professor in the Department of Computer Science at Florida State University and recently took a research sabbatical at Los Alamos National Laboratory. His research interests include parallel and distributed systems, interconnection networks, communication optimizations, and networking. He obtained his BS and MS degrees in computer science from Shanghai Jiaotong University in 1989 and 1992, respectively. He earned his PhD degree in computer science from the University of Pittsburgh in 1998. He publishes extensively on interconnection networks and communication-library implementation and optimizations.

The Self-Tuned Adaptive Routines for Message Passing Interface (STAR-MPI) software package that he and his students developed has been incorporated into the software stack of IBM's Blue Gene/P supercomputer. Professor Yuan is currently serving on the editorial boards of several international journals. He has also served as the program chair and vice chair for several international conferences and workshops, such as the International Conference on Parallel Processing (ICPP) and the Institute of Electrical and Electronics Engineers (IEEE) International Conference on High Performance Computing (HiPC), and as a program committee member for many international conferences and workshops. He is a senior member of both the Association for Computing Machinery (ACM) and IEEE.

Michael Lang is the team leader of the Ultrascale Systems Research at Los Alamos National Laboratory. His research interests include distributed services, performance of large-scale systems, operating-system and run-time issues for supercomputers, and interconnects for large-scale systems. He has published work on the application-specific optimization of routing on InfiniBand interconnects for large-scale systems. Notably, this algorithm is currently included in OpenSM in the OpenFabrics software stack. Lang was formerly

a member of Los Alamos National Laboratory's Performance and Architecture team, involved in performance analysis of new large-scale systems for the US Department of Energy. He received a BS in computer engineering and an MS in electrical engineering in 1988 and 1993 respectively, both from the University of New Mexico.

## References

[1] Kothe DB, Baumgardner JR, Cerutti JH, Daly BJ, Holian KS, Kober EM, Mosso SJ, Painter JW, Smith RD, Torrey MD. "PAGOSA: A massively parallel, multi-material hydrodynamics model for three-dimensional high-speed flow and high-rate material deformation. In: Tentner A, editor. *High Performance Computing Symposium 1993: Grand Challenges in Computer Simulation* (Proceedings of the 1993 Simulation Multiconference on the High Performance Computing Symposium; Mar 29–Apr 1, 1993, Arlington, VA). San Diego (CA): Society for Computer Simulation; 1993. p. 9–14. ISBN: 978-1565550520.

- [2] Lamport L. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM*. 1978;21(7):558–565. doi: 10.1145/359545.359563.
- [3] Meuer H, Strohmaier E, Dongarra J, Simon H. TOP500 Supercomputer Sites: June 2013 [accessed 2013 Aug 2]. Available at: http://www.top500.org/lists/2013/06.
- [4] Öhring SR, Ibel M, Das SK, Mohan J K. "On generalized fat trees." In: *Proceedings of the 9th International Parallel Processing Symposium*; Apr 1995, Santa Barbara, (CA). p. 37–44. doi: 10.1109/IPPS.1995.395911.